# Introduction

## Objectives

* The framework keeps the domain very clean (the domain objects do not have to implement INotifyPropertyChanged or INotifyCollectionChanged).
* Much less code for the view models because most things can be configured declaratively.
* Maintainability.
* Testability of application logic code.

## Application Architecture

The following diagram depicts the basic elements of a simple employee management application build with the Inspiring MVVM Framework. This is basically an implementation of the well-known MVVM design pattern.



### Screens and Screen Conductors

A screen is some part of the user interface that is displayed to the user (e.g. an employee form) and that has a lifecycle (it is opened, activated, deactivated and finally closed).

Screens are usually responsible for loading and saving the source objects by (using the domain layer), managing the Unit of Work, and controlling the screen lifecycle. The screen itself should not expose bindable data but usually creates and exposes a single view model (which may of course be a view model hierarchy).

A screen conductor is responsible for managing a collection of screens. The corresponding view is determines how the managed screens are displayed (e.g. as tabs, as MDI windows or on top of each other). The screen conductor tracks the currently active view and forwards the lifecycle events (like activate) to the screens it contains.

### View Models

View models implement the application logic and the application workflow. View models usually wrap one or more domain objects and transform them into a form that is easily consumable by the view.

# The Inspiring MVVM Framework

The following diagram depicts the major areas of the Inspiring MVVM framework:



## Application startup

The startup process is controlled by the MvvmApplication class. The App class of your application should derive from this class.

## View location

Views are usually displayed in elements that are ControlControls. With the Inspiring MVVM framework you always display Screens and not Views. Because Screens are UI independent, it is necessary to find the corresponding View before the Screen can be displayed.

This is done by setting/binding the View.Model attached property on the ContentControl to a Screen. Every time its value is set, the following process applies:

1. Check if the Content property of the ContentControl contains already a view that implements IView<TScreen>. If so, just update its IView<TScreen>.Model property and return.
2. If not, use the IServiceLocator to create a new object that implements IView<TScreen> where TScreen is the type of the assigned screen object or a base type of it.
3. Set the Model property of the resolved view to the screen.
4. Set the Content property of the ContentControl to the resolved view.

## The IServiceLocator

The Inspiring MVVM framework requires some mechanism of dependency injection/service location to create new view models for example. The framework does not prescribe a certain DI container but only requires you to implement an adapter that implements the very simple IServiceLocator and forwards the calls to your DI container of choice.

# View Models

A view model definition always consists of two parts:

* The VMDescriptor (EmployeeVMDescriptor) defines metadata (VMPropertyDescriptor objects) for all properties that a concrete ViewModel holds and for the view model itself.
* The ViewModel class (EmployeeVM) contains implements the application logic required for a part of the UI. All instances of a given view model class share the same VMDescriptor instance. The VMDescriptor instance is created and configured by the concrete ViewModel once in its static constructor and stored in the public static field ClassDescriptor.

Example:

public class Person {

public string Forename { get; set; }

public string Surname { get; set; }

}

public sealed class EmployeeVM :

ViewModel<EmployeeVMDescriptor>,

IHasSourceObject<Person> {

public static readonly EmployeeVMDescriptor ClassDescriptor = VMDescriptorBuilder

.OfType<EmployeeVMDescriptor>()

.For<EmployeeVM>()

.WithProperties((d, c) => {

var p = c.GetPropertyBuilder(x => x. Source);

d.FirstName = p.Property.MapsTo(x => x.Forename);

d.LastName = p.Property.MapsTo(x => x.Surname);

})

.Build();

public EmployeeVM()

: base(ClassDescriptor) {

}

public Person Source { get; set; }

}

public sealed class EmployeeVMDescriptor : VMDescriptor {

public IVMPropertyDescriptor<string> FirstName { get; set; }

public IVMPropertyDescriptor<string> LastName { get; set; }

}

The following diagram depicts how the elements play together:



## Building the descriptor

The VMDescriptorBuilder is used to create and configure a descriptor for a ViewModel.

public static readonly EmployeeVMDescriptor ClassDescriptor = VMDescriptorBuilder

.OfType<EmployeeVMDescriptor>()

.For<EmployeeVM>()

.WithProperties((d, b) => {

var p = b.GetPropertyBuilder(x => x.PersonSource);

d.FirstName = p.Property.MapsTo(x => x.Forename);

d.LastName = p.Property.MapsTo(x => x.Surname);

})

.Build();

The VMDescriptorBuilder has the following configuration options (implemented as fluent interface methods):

|  |  |
| --- | --- |
| Property type |  |
| Inherits | Inherits from the specified descriptor. |
| OfType | Specifies the concrete VMDescriptor class. |
| For | Specifies for which concrete ViewModel the descriptor should be configured. |
| WithDependencies | Specifies that a change of one property should refresh another. |
| WithValidators | Specifies how the view model should be validated |
| WithBehaviors | Declaratively enables and configures behaviors. |
| Build | Returns the configured descriptor. |

## Defining view model properties

|  |  |
| --- | --- |
| Property type |  |
| Mapped | The property gets/sets the value of property on the source object. |
| Delegated | The property calls a delegate to get/set when its value is get/set. |
| Simple | The value of property is stored by the view model and can be get/set like a normal CLR property. |

## Accessing VM properties

A VM property may have multiple values:

|  |  |
| --- | --- |
| Value stage | Description |
| Display value | The value of the property as it is displayed by the view.  The display value is always of type object because it may be of a different type than the VM property (e.g. a number might be a string in the view which is then converted to an int by the view model). This value is usually bound to a UI element via WPF data binding. |
| Unvalidated value | The value of the property after all display value validations and the conversion to the target type of the property succeeded but before the property validations are checked.  You usually want to work with this value in your view model code. |
| Validated value | The value of the property after all validations succeeded. |
| Source value | The value of the mapped target property on the source object (if it is mapped to a source object) or the value returned by the getter delegate (if it is a delegated property).  The source value can only be different from the validated value if the VM property is disconnected from its source. |

The following diagram illustrates the different value stages:



An example: the property Salary is of type decimal and has defined a validator that rejects negative values. The value of the property on the source object is 500. The following table lists the value that the property has if its display value is set to the value in the first column.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| New Display value | Case description | Display value | Unvalidated value | Validated value | Source value |
| Way too little | The conversion fails. | Way too little | 500 | 500 | 500 |
| -300 | The conversion succeeds but the property validation fails. | -300 | -300 | 500 | 500 |
| 700 | The conversion and validation succeed. | 700 | 700 | 700 | 700 |

### Accessing VM properties in code

The different values of a property can be accessed with the GetDisplayValue, GetValue and GetValidatedValue and set with the SetDisplayValue and SetValue methods of the ViewModel class.

If you work with the value of a property in your view model or code behind code, it is recommended that you write a simple wrapper property:

private string FirstName {

get { return GetValue(Descriptor.FirstName); }

set { SetValue(Descriptor.FirstName, value); } // only if needed

}

This makes it clear which properties are only bound to the UI (no wrapper) and which are accessed in code (with wrapper).

Please note that you do not need to write such wrapper properties you do only bind the property in to the view!

### Binding VM properties

Because the ViewModel implements the System.ComponentModel.ICustomTypeDescriptor interface you do not have to define wrapper properties for all your VM properties.

Before the WPF binding system checks if the type of the current DataContext object has a CLR property with the bound property name, it checks if the DataContext object implements the ICustomTypeDescriptor interface. If this is the case it calls GetProperties on the interface and checks if the returned collection contains a PropertyDescriptor with a Name equal to the bound member. If this is the case, the binding system calls PropertyDescriptor.GetValue and PropertyDescriptor.SetValue to get or set the bound property. This is the reason why you can bind to properties that do not really exist on your view model class. The following diagram illustrates this:



A VM property can be bound in the same way as any normal property but it is recommended that you use the ViewBinder to bind the properties in a type safe way (see corresponding chapter).

## Child VMs

Are mapped with v.VM.Of<>, v.VM.Wraps, v.VM.DelegatesTo.

[TBD]

## Child VM collections

Are mapped with v.Collection.Of<>, v.Collection.Wraps, v.Collection.PopulatedWith.

[TBD]

## Important ViewModel methods

|  |  |
| --- | --- |
| Method | Description/Usage |
| Refresh(property) | Refreshes the value of a property.  This method should be called, if the value source of a property has changed by some way other than changing the VM property.  If a property becomes out of date because another VM property has changed, it is preferable to indicate such dependencies declaratively (feature is coming soon).  This method should be used very sparingly. |
| Revalidate(scope, mode) | Call this method to revalidate a view model manually.  Validation happens automatically in most cases. The most common case is the Initialize method of a root view model.  This method should be used sparingly. |
| CopyFromSource/ CopyToSource | This method is only relevant if the view model is a disconnected view model (see corresponding chapter). |
| GetValidationState | Returns the current validation state of the view model. |

## Validation

**Validation** means the process of validating a property or view model. A **validator** in contrast means the method, object or class that is capable of doing the validation.

There are two types of validations:

**Property validations** are performed when the value of a property is about to change. The validation checks the new value the property is going to assume. If all property validations of a property succeed, the new value is written to the source object. Otherwise the new value is cached as the unvalidated value of the property, but the source object remains unchanged.

**View model validations** are performed every time the view model it targets or any of its descanting view models has changed (a property has changed or the validation state has changed). It is important to note that view model validations are performed after a property has changed which means they do not prevent the modification of the source objects.

The **validation state** is the result of a validation. It holds all validation errors and the overall result (valid/invalid). Each property and the view model have a validation state.

### Defining view model validators

[TBD]

### Validation scope

The validation scope specifies what part of the VM hierarchy is revalidated.

|  |  |
| --- | --- |
| Validation scope | Description |
| Self only | Only the current VM itself is validated. All validations that ancestors defined for this VM are also performed. |
| Full sub tree | The current VM and all its descendants are revalidated. |
| Self and validated children | The current VM is revalidated and all children, for which the current VM has defined validations. |

### Validation mode

The validation mode defines how revalidation is performed.

|  |  |
| --- | --- |
| Validation mode | Description |
| Commit valid value | A property may have invalid pre-conversion and pre-validation values. If you revalidate with this option the VM revalidates the invalid values write them to the source object it the have become valid. |
| Discard invalid value | The pre-conversion and pre-validation values of the property are discarded and the post-validation is revalidated. |

### Validation arguments

|  |  |
| --- | --- |
| Value type | Description |
| Owner VM | The VM whose descriptor has defined the validator. |
| Target VM | The VM that should be validated. |
| Target path | A VM instance path that contains all VMs from the owner VM to the target VM (including both). |
| Target property | The property that should be validated. This property is only valid for property validations. |
| Changed VM | The VM whose change has triggered the validation. In case of a property definition it contains the VM that is about to change. |
| Changed path | A VM instance path that contains all VMs from the target VM to the changed VM (including both). |
| Changed property | The property that has changed (if any). This property is only valid for view model validations. |
| Errors | Holds all validation errors that occur in the current validation process. |

Example 1: A validator defined on the createViews VM is called because a property of it has changed.



Example 2: A validator defined on the list VM is called because the createViews VM has changed.



# View Models (not entirely up to date…)

VM Properties

|  |  |
| --- | --- |
| Value type | Description |
| Mapped | A mapped property reads and sets the value of a source object property.  Every time the VM property is accessed on a VM the getter of mapped property of the VMs source object is called. Every time the VM property is set on a VM the setter of the mapped property of the VMs source object is called.  A mapped property may also map to a complex property path of the source object, such as Invoice.Customer.Address.Street. In this case all properties are read in sequence and the value of the last property is returned or set on the source object. |
| Delegated | A delegated property calls a delegate when VM property is read or set. |
| Simple | A simple property stores its value in the VM. It is similar to a normal get/set property but enhanced with all VM property features. |

### IVMPropertyFactoryProvider and IVMPropertyFactory

The IVMPropertyFactoryProvider creates IVMPropertyFactory objects that can be used to create VM property instances.

By default the source object for VM properties created by a property factory is the VM itself. It is also possible to create property factories which create VM properties that have a source object that is referenced by the VM, such as <VM>.InvoiceSource.Address.

# Framework Developer Documentation

|  |  |
| --- | --- |
| Source folder | Content |
| Core |  |
| Collections | The default IVMCollection implementation, all collection behaviors and property behaviors that are only useful for collection properties (excluding any behavior that is concerned with validation). |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

## Behaviors

### Motivation

View models often have to implement several cross-cutting concerns such as validation, synchronization with the domain model, change notification, authorization and inter-property dependencies.

Implementing all this concerns in one class is not very practical. This is further complicated because there are many different combinations of these concerns varying between VM properties and between view models. The result is often lots of code repetition and code that is complex and hard to maintain.

To overcome these problems most of the functionality in this framework is implemented as behaviors.

### Behaviors

View models and VM properties provide several operations on them such as GetValue, GetValidationState, Revalidate, UpdateFromSource, etc.

These operations are not implemented by the view models and properties themselves but are delegated to a behavior chain. Each operation is defined in a behavior interface (the terms object, class and interface refer to the .NET concepts in this section). A behavior chain is a linked list of behavior objects. Each behavior object (or more exactly its class) may implement one or more behavior interfaces. Each VM property and the view model have their own behavior chain.

When an operation is for a view model or VM property is invoked, the first behavior object in the behavior chain (of the view model or VM property) that implements the behavior interface on which the requested operation is defined is invoked.

A behavior interface method implementation may (and usually should) call the next behavior in the chain that also implements the behavior interface (if there is such an object). It can execute arbitrary code before or after it forwards the operation to the next behavior. It may also call operations on other behavior interfaces implemented by behavior objects that come anywhere after the current object in the chain.

This means that a single operation can be processed by multiple behavior objects/classes. This pattern is based on the Chain-of-responsibility design pattern.

[TBD: graphical illustrations, examples]

### Behavior configuration

Different VMs and VM properties may have different behaviors. These behaviors are determined by the VMDescriptionBuilder (methods like WithProperties, WithValidations, WithBehaviors may all add or configure behaviors).

What behaviors a chain may have and the order of them is determined by a BehaviorChainTemplate. Because there are main BehaviorChainTemplates, all available templates are managed the BehaviorChainTemplateRegistry. A template is identified by a template key (an arbitrary object).

The VMDescriptorBuilder creates BehaviorChainConfigurations based on the templates which are used to hold transient behavior settings while the VM descriptor is constructed. The BehaviorChainConfigurations of the VM and all VM properties are managed by a VMDescriptorConfiguration object.

At the end the VMDescriptorBuilder creates the concrete behavior chains from the BehaviorChainConfigurations.

## Validation

### Validation context

The **validation context** holds context that is shared during a single validation process. It holds various validator contexts and the revalidation queue.

### Revalidation queue

If a validator validates other VMs in addition to its target VM and notices that their validation state may have changed, it can add them to the **revalidation queue**. All VMs in the revalidation queue are validated sometime after the current validator invocation has finished.

### Validator context

If a parent VM type P defines a validator for an ancestor VM type A that validator may be called many times during a single validation process when following conditions apply:

1. There is a collection relation anywhere between P and A.
2. Either a full validation is performed (meaning that every item of the collection is validated) or a validator defined for A adds other instances of A to the revalidation queue.

In this case the same **validator context** is passed to all invocations of the validator that are performed for a certain instance of P. Note that validator invocations of two different instances of P do not share the same validator context.

### The validation process

#### Revalidate view model

**Main success scenario** (validation mode self only):

1. The VM revalidates all properties with the specified validation mode.
2. The VM performs view model validations.

**Extensions:**

1a. The revalidation scope was full sub tree:

1a1. Before validating the properties, revalidate the children of the VM with validation scope full sub tree and the given validation mode.

1b. The validation scope was self and validated children:

1b1. Before validating the properties, revalidate all children for which the current VM has validations defined.

#### Revalidate property (commit valid values)

**Main success scenario:**

1. The VM gets the pre-conversion value of the property.
2. The VM sets the pre-conversion value of the property, which performs the property validation.

#### Revalidate property (discard invalid values)

**Main success scenario:**

1. The VM clears the pre-conversion and pre-validation cache.
2. The VM performs a property validation on the post-validation value of the property.

#### Perform view model validations

**Main success scenario:**

1. The VM raises the view model validating event.
2. The VM forwards the event to all ancestor VMs.
3. The VM or any ancestor VM may handle the event and add validation errors to the validation state.
4. The VM saves the new validation state.
5. The VM compares the new validation state to the previous validation state and raises a validation state changed event if it has changed.

#### Perform property validation

**Main success scenario:**

1. The VM raises the property validating event.
2. The VM forwards the event to all ancestor VMs.
3. The VM or any ancestor VM may handle the event and add validation errors to the validation state.
4. The VM saves the new validation state.
5. The VM compares the new validation state to the previous validation state and raises a validation state changed event if it has changed.

## Notes

Manual updates

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Property type | UpdateFromSource | | UpdateSource | |
|  | **Pass-through** | **Disconnected** | **Pass-through** | **Disconnected** |
| **Property** |  |  |  |  |
| Mapped | Notify property change | Refresh source & Notify property change | None | Update source |
| Delegated | None |
| Local | None | None | None | None |
| **VM Property** |  |  |  |  |
| Wrapping | Update VM | Refresh source & update VM | None | Update source |
| Delegated | Call getter | Call getter | None | ? |
| Local | None | None | None | None |
| **Collection** |  |  |  |  |
| Wrapping | Repopulate | Refresh source & repopulate | None | Update source collection\* |
| Populated | Repopulate | Repopulate | None | None |
| Local | None | None | None | None |

\* Not implemented

Notes:

~~Revalidate with mode Discard Invalid Values for view model properties that are currently invalid recreate the child VM. Validations on view model properties are only useful for view model properties that are actually set from outside.~~

Revalidate with mode Discard Invalid Values for view model properties that are currently invalid resets the property valid to the last valid value (it does not create a new VM for the current source value).

Refresh

|  |  |
| --- | --- |
| Property type | Results |
| **Property** |  |
| Mapped |  |
| Delegated |  |
| Local |  |
| **VM Property** |  |
| Wrapping |  |
| Delegated |  |
| Local |  |
| **Collection** |  |
| Wrapping |  |
| Populated |  |
| Local |  |